A Revised Study of the Pull-based Software Development Model

Alexandru Balan
Delft University of Technology
Delft, Netherlands
A.Balan@student.tudelft.nl

Andra-Denis Ionescu
Delft University of Technology
Delft, Netherlands
A.Ionescu-3@student.tudelft.nl

Jeffrey Sirocki
Delft University of Technology
Delft, Netherlands
J.A.Sirocki@student.tudelft.nl

ABSTRACT

It is undeniable that the pull-based software development model is an established approach to build sustainable open-source projects. A study in 2013 began an investigation into the adoption and use of the model; we provide an extension upon this work. Our project goal is to understand how developers use pull requests in DVCS systems. We perform an identical analysis to the previous study, looking at two datasets containing 829 projects from GitHub. Our work, presents a thorough comparison between our findings and the previous study, examining how the model has evolved over the last five years. We show that pull requests grew increasingly popular due to the rising popularity of Github. We find that pull requests offer faster turnaround, yet, teams take longer to process external contributions and closed pull requests. We identify a small array of factors as truly influencing the merging outcome and find the discussion of a pull request highly indicative of the merge time.

Keywords

pull-based development; pull request; github; distributed software development

1. INTRODUCTION

Software Engineering practices evolve through the continual advancement of software tooling. Often times, new development tools bring a wave of enthusiasm and improvement. With the introduction of distributed version control systems, such paradigms started to evolve. An example is the pull-based software development model. This model separates development from the integration effort by allowing external contributors to submit changes. Such contributions improve code quality and collaboration. Once submitted, the core team evaluates and decides whether the project will merge the changes or discard them. GitHub is one popular example of a code hosting website that incorporates this new model into its core functionalities. Many parties have adopted this model, but, the majority, in 2013, [1] did not. There is a disparity in how project teams conduct their development. Some view the pull-based model as a revolution for open-source development while others prefer a standard push model.

A previous study [1], which our work extends, provides insight on developer preference and the pull-based development model. The work discusses the popularity and lifecycle characteristics of the pull-request development model, looking particularly at which factors are influential. The study is the first to look at pull requests with substantial data, however, its results, are out of date according to GitHub's 2017 Octoverse report[2]. The report indicates that the total number of pull requests increased from 9.2 million in 2013, the year of the previous study, to over 130 million in October 2017. Another look to see whether the characteristics and properties changed would be indicative of how the pull-based development model evolved in the last four years. Because software trends change quickly and the development model is more mature, we extend this previous study to reveal different developer behavior and show pull request *merge time* and *acceptance* trends.

The research questions we are answering now are exactly the same as those proposed in 2013, except we do not answer the fourth question [1]:

RQ1: How popular is the pull-based development model compared to 2013?
RQ2: What are the life cycle characteristics of pull requests?
RQ3: What factors affect the decision and the time required to merge a pull request?

Our study extends the work of the previous authors through the use of updated datasets, enhanced tools, and improved algorithm. We extract data via the GitHub API and GHTorrent project, a mirror of the GitHub API and then answer each question.

The first research question assesses whether the GitHub community acknowledges the benefits of the pull-based development model. We verify if the adoption model scales to a growing platform and highlight existing trends within the developer community. Using GHTorrent, we determine the popularity of pull requests in repositories from the past two and a half years as described in Section 4.1.

The second and third research questions use a more in-depth approach because the extraction of lifecycle and merge related characteristics require complex operations and algorithms. To provide developers a better understanding of pull-based development habits, we investigate two datasets containing pull requests between September 2016 and September 2017.

The first dataset contains repositories from GHTorrent and the second dataset contains repositories from the RepoReaper's project. For the first dataset, we select 749 projects that meet the criteria specified in Section 4.1. For the second, we pick 80 repositories that fit the RepoReaper's dimension criteria as explained in Section 4.2. Together, the data sets provide an updated basis for our study that best represents the GitHub community.

We analyze a range of projects developed in five languages (Ruby, Python, Java, Javascript, and Scala, containing over 320,000 pull requests). To see which factors affect pull request lifetime and merging, we apply pattern recognition techniques and compare the GHTorrent and RepoReapers datasets for accuracy. By answering these questions, we show how developers can begin to build a set of guidelines to get better manage and improve efficiency within their projects.

In this study, we will not argue *RQ4: Why are some pull-requests not merged?* The previous authors looked into the discussions of 350 unmerged pull-requests and classified the reasons the developers closed each. We decided that this is unfeasible and a task that falls outside of the scope of this project because without the use of natural language processing algorithms, the manual processing 350 or more requests is too expensive. Thus, we are focusing our efforts on the first three research questions, the core of the previous study.

2. *BACKGROUND*

New software tooling often brings new software paradigms. In our study, we focus on pull requests within Git[3], the most popular DVCS, distributed version control system. DVCS are a leg up on CVS, centralized version control, because developers can (i) work offline on local copies without connection to a network, (ii) commit, view history, and revert changes, and (iii) communicate only when necessary to share changes amongst peers. Git's team oriented characteristics have led to incredible growth of Git and GitHub[4, 8]. In the last decade, GitHub, the most popular platform for Git, has grown to almost 34 million users, 62 million repositories, and over 130 million pull-requests[2]. Furthermore, October 2017 trends on Google[7] show that Git held 83% of all web search interest when compared with Mercurial[5] 6% and SVN 9%[6]. To understand the adoption of Git, we look deeper into DVCS systems and pull requests.

In shared repository development, developers add features through a pull request. A pull request, one of Git's trademark features, specifies a local branch for the system to merge with a branch in the main repository. If the request is satisfactory then the main repository pulls the proposed changes, and, if deemed unsatisfactory, revisions may be necessary. When the core team requires revisions, the external contributor makes the desired changes and submits them for a second review. Once the developer addresses the team's concerns and, if there are no merge conflicts, then the request goes through to the main repository. On average many pull requests are successful, however, to understand why some pull requests are not successful, we must consider the contributing factors.

2.1. PULL REQUESTS ON GITHUB

Pull requests, in their simplest form, are a way for a developer to notify team members that they have completed a feature. Pull requests, however, are more than a notification. They are a process to discuss, review, and document proposed features [11]. All discussion and commit activity in projects is often tracked inside of pull requests [9] as shown in Figure 2. Thus, they are an important tool for mining repository history.

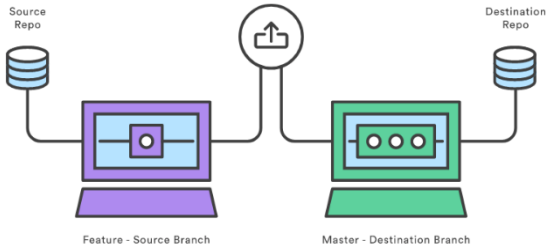

Figure 2: A pull request that asks to merge a feature branch into the official master branch [9].

When a developer files a pull request they provide four pieces of information: the source repository, the source branch, the destination repository, and the destination branch. These values depend on the collaboration workflow specified by the team [12]. Additionally, the information recorded depends on how the core team merges the pull request.

When the core team deems a pull request satisfactory and the review is over, the system merges a pull request in one of three ways [13]. The first method, through GitHub, automatically merges the applicable commits, records the authorship, and stores the history information of the merge event. The second method, using Git merge, involves merges that the team can not complete through the GitHub interface and muse use of Git utilities. In this case, the main difference is there is manual effort required and the author is different; the author is the person who merged the commit. The last method, committing a patch, involves any merger where the commits go to an upstream branch. In each case, the system merges two branches and stores information on pull requests.

Considering these merge characteristics and the variety of information incorporated within pull requests, we aim to assess the history of pull-based development on GitHub. Moreover, developers use pull requests as an analysis and design tool, indicator of progress, and as a reliable source of documentation. To uncover to what extent developers use pull requests for these purposes, as well as, understand how pull requests perform in meeting the needs of developers, we pull insight from pull-request data over a one year period.

3. *METHODOLOGY*

The main focus of this paper is to understand how developers currently use Git pull requests within large open source projects as an enabler of organization and collaboration. The answer to our research questions may provide insight into the current pull request trends and show improvements for pull requests. To answer our questions, we sequentially describe how we collect our data, use quantitative data analysis to understand the problem, and, highlight important findings. Below, we present each research question.

RQ1: *How popular is the pull based development model compared to 2013?*

To answer the first research question, we generate and analyze statistics that express the use of pull requests inside GitHub.

The initial paper analysed data for a period of one and a half years, from February 2012 to August 2013. To extend the previous paper's work, we analyse recent data and compare it with the pull-based development model data from 2013 We chose a timeframe of two and a half years between March 2015 and September 2017. We did not start from August 2013 until present, because our interest is in recent data and because there is not enough time to collect and process all available data.

To observe these statistics over time, we divided the timeframe between March 2015 and September 2017 into 5 distinct datasets, each containing data for a period of 6 months. If we had used the whole period to analyse the evolution, we could not have observed an important growth. And, if we had used smaller timeframes, we would not conform to the time allocated for this research.

We are analyzing the total number of commits and pull requests for each timeframe to get an overview of how popular Github is. We calculate the percentage of pull request out of the total number of commits to assess the popularity of the model in general. Moreover, we look at the number of active projects and how many of them actually use pull requests. Computing the percentage of projects that use pull request will give us the best metric in assessing how popular pull requests are. To assess whether a change occurred in the last couple of years, we compare the popularity with the previous paper.

RQ2: *What are the life cycle characteristics of pull requests?*

RQ3: *What factors affect the decision and the time required to merge a pull request?*

To answer the second and third research questions, we use two datasets obtained through two different approaches. The first dataset contains repositories we extracted from GHTorrent on a selective criteria explained in Section 4.1. The second dataset includes repositories from the Reporeapers project that satisfy the RepoReapers

dimensional criteria explained in Section 4.2. In these datasets, we analyze over 320,000 pull requests.

To determine the lifecycle characteristics, we study the total number of pull requests in each repository then determine the percentage of pull requests that fall into specific quartiles (hour, day, more than a day). We group and count pull requests based on these points to best split the available data into bins of nearly equal size. Using the data from the bins, we determine the average amount of time it takes to merge a pull request looking at contributing factors such as discussion and pull request size to answer RQ2 in Section 6.

Using these two sources of data, we then apply, in Section 7, six machine learning algorithms to determine which factors affect both the decision and the time required to merge a pull request. Using pattern recognition techniques, we retrieve the dominant features through two main steps.

The first step is running six classification algorithms including Random Forests (randomforest), Logistic Regression, a binary classification for the merge decision task (logregr), Naive Bayes (naivebayes), Support Vector Machines (svm), decision trees (dtree), and AdaBoost with decision trees (adaboost). We select these algorithms because they work well with clustering large datasets and in prediction models. The previous study deployed these machine learning algorithms, proved their capabilities, and additionally, since 2013, improved their implementations of the algorithms. Considering the improvements, we plan to use their tuned algorithms without further modification to report the top three features classifying merge decision and merge time.

Our second step to answer RQ3, takes the top classifiers from the first step and applies a classifier-ranking process to validate their importance. We check the classification performance through two metrics: Accuracy (ACC) and Area Under the Curve (AUC). We use a 10-fold cross-validation process to analyze the predictor features, starting with the best feature. In each iteration, we analyze a specific feature then add to the model the next most important feature. In this process, we compare each feature and rank their importance. When a feature surpasses a threshold, we report that it can predict the classification outcome with reasonable certainty, and regard it as an important feature. This two-step classification process provides us with a validated set of ranked features that affect merge decision and merge time, thus answering RQ3.

4. DATA

In this study, we use data from the GitHub API; more specifically, we use GHTorrent [16], an off-line mirror of the data on the GitHub API. The GitHub API offers both static and real-time data. The static view contains the current state and number of repositories on GitHub. The real-time data includes a stream of data events such as forks, pull requests, push events, and other actions occurring in real-time. To get this data, GHTorrent applies a recursive dependency-based search [14], starting from the references of the root entities, that yields all data offered via the GitHub API. The data begins in an unprocessed format in a MongoDB database and the system extracts the metadata and stores it in MySQL.

4.1 GHTorrent Project Selection

Software repositories contain valuable information about the code, authors, contributors and other metadata. Researchers can use this information in retrospective analysis to look into the evolution and growth of software products. Since open source software repositories are accessible, we have plenty of data to mine.

To make our analysis comprehensive, we filter the personal projects, toy projects, and inactive projects to focus on relevant repositories. In particular, the projects we are taking into consideration are those that have been active in the last year (had at least one commit), have a ratio of pull request to total number of commits of at least 50%, contain at least 200 pull requests throughout its history and are not forks of other repositories. The timeframe used in this case is smaller than the one used for answering RQ1 because it would need more processing time without adding significant benefits.

These criteria led us to 1133 projects (396 JavaScript, 258 Ruby, 247 Python, 211 Java, and 21 Scala). We additionally use a pull request extraction tool [1], which filtered our selection to 749 projects (255 Javascript, 192 Python, 159 Java, 128 Ruby and 15 Scala). This change in numbers is due to the fact that some pull request no longer contain all the required metadata due to various reasons, such as deletion, data privacy or the impossibility to detect merges. The reason we chose these specific programming languages is to maintain compatibility with the the tools used to mine the repositories. Together, the projects offer a representative distribution of programming languages and pull requests on GitHub over a one year period. To confirm our filtering methods, we investigate another set of repositories from

RepoRepears, projects selected for research purposes.

4.2 RepoReapers Project Selection

The curiosity to perform the same analysis on a proven dataset lead our team to using RepoReapers. We compare a tested dataset and untested dataset to find interesting outcomes.

Reapers is a framework that extracts all the open source software projects and classifies them based on a criteria of eight dimensions: Community, Continuous integration, Documentation, History, Issues, License and Unit testing[10]. The RepoReapers identify repositories that meet all eight dimensions with the aim of providing a set of repositories for research purposes.

The classification aims to separate engineered software projects and those that are not. An engineering project is one that popular engineering organizations develop and, moreover, is useful to other developers outside the internal team. To extract such projects, the RepoReapers team utilized two algorithms, *random forests* and *score-based* to perform separation.

In our study, we work with the list from RepoReapers and perform filtering using a set criteria. To ensure an acceptable project selection, we accept all projects marked by their classifier algorithms and only consider projects that contain at least 100,000 lines of code. Additionally, the presence of unit testing, continuous integration, and having at least 10 stars are other signs of software quality that have to be present in the filtered repositories. Because we use the same tools to extract pull requests from GHTorrent, we have only selected repositories that use compatible programming languages. Our final dataset includes 80 projects (*41 Java, 25 Python, 12 Ruby, 2 Javascript*).

5. POPULARITY OF PULL-BASED DEVELOPMENT

We have found that Github has grown more and more popular, leading to an increasing number of commits and pull requests. In Figure 3, we show Github between March 2015 to September 2017 registered a growth of 85% in the number of commits and a growth of 105% in the number of pull requests. The adoption seems more obvious when indicating the difference between the average number of pull requests per month; In January 2013 developers submitted 150,000 pull requests, which is more than 5 times smaller than in September 2017. Since the previous study, the usage of pull requests has grown enough to warrant study.

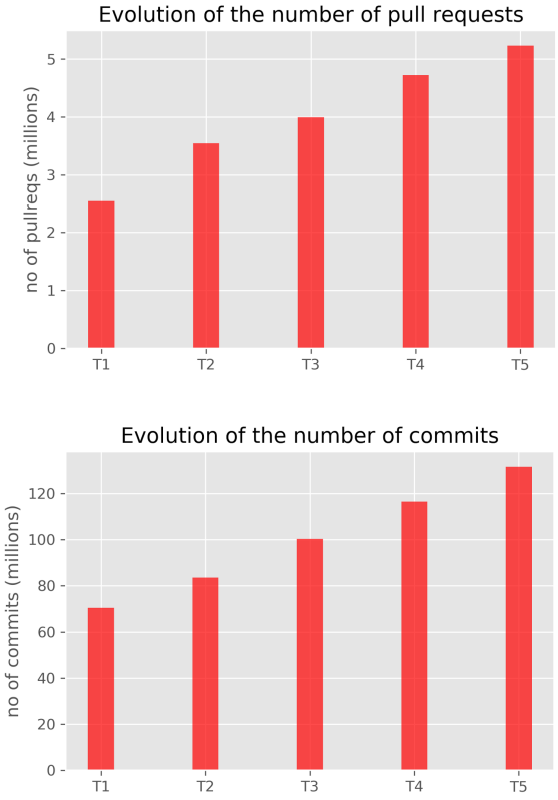

Figure 3. Number of commits (b) and pull requests (a) for each timeframe (T1: March 2015 - September 2015, ..., T5: March 2017 - September 2017)

In these figures, we see the number of both pull requests and commits increased. Since, our interest is to understand the popularity of pull requests, we use this data to compare the number of commits to pull requests.

In Figure 4, the percentage of pull requests out of the total number of commits fluctuates around 4%; registering its highest mark of 4.2% in the second timeframe of 2015. This indicates the popularity of pull requests to commits is steady on the Github platform. This might mean the growing popularity of Github, indicated by number of commits, yields an increase in number of pull requests.

As of September 2016, Github reported to have over 19.4 million active repositories. We found that between March 2016 and September 2016, only 7.2 million repositories performed at least one commit, out of which only 5.9 million are original repositories. This difference in numbers, we found, is interesting and misleading. It might result from a different definition of active repositories or because

Github has chosen a bigger timeframe to assess whether repositories are active.

Regardless, by analyzing the active projects (performed at least one commit in the last 6 months), we see the same ascending trend in Figure 5 (right), as in the general analysis.

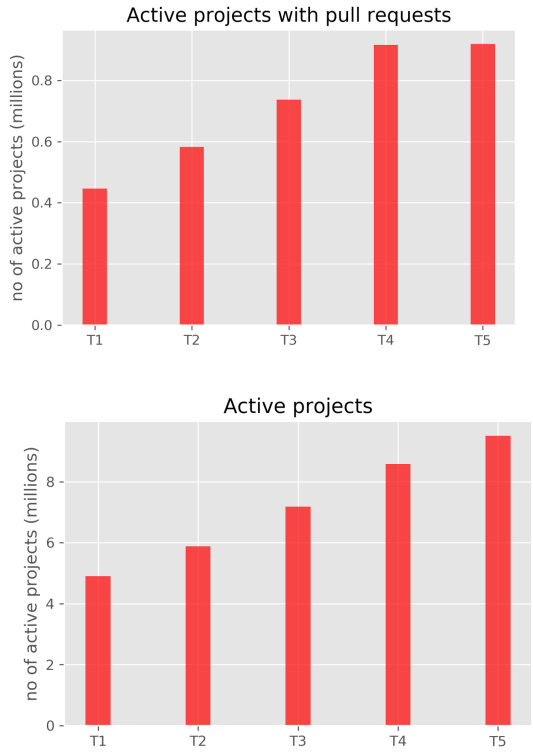

Figure 5. Number of active projects (b) and active projects that use pull requests (a) for each timeframe

We see active pull request development model projects, as shown in Figure 5 (left), follow a similar growth pattern, except for some leveling off in the last timeframe. The popularity of pull requests against commits reflects a similar leveling off, as shown in Figure 6. Here, the percentage of projects that use pull requests reveal a general positive trend, except for the last timeframe too, suggesting that the ratio of pull request projects to pull request use is holding steady. Furthermore, without future data to compose an additional timeframe, it is hard to assess whether these pull request trends will remain the same, decline, or if it is a usage anomaly. Such a phenomenon might be a starting point for further research.

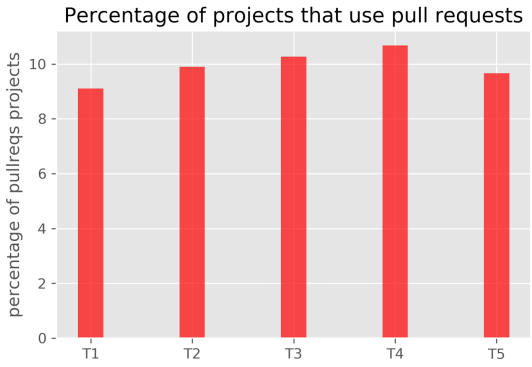

Figure 6. Percentage of projects that use pull requests

In the case of the more general statistics, the difference in numbers between what the previous authors measured in 2013 and the present is significant. In January and February of 2013, 42,400 repositories used the pull request development model, while in August and September 2017, 350,934 repositories performed at least one pull requests. To conclude, both approaches to this research question reveal, that during the targeted period, the popularity of the pull request development increased, most likely due to constant growth of Github as a platform.

6. PULL REQUEST LIFECYCLE CHARACTERISTICS

Lifetime of pull requests. After a developer submits a pull request and its team finishes their review, the pull request can be in one of two states: merged or closed. In our dataset, the projects merged 79.69% of 320,000 pull requests, while the rest are not. Compared to the previous study, the percentage of merged pull requests has **increased by 5.2%**. This might mean that developers have a better understanding of the pull based software development model now as compared to 4 years ago. An extended explanation for this increase falls outside the scope of this paper as it requires further processing and might be a starting point for future research. Applying the same process to the reporeapers dataset, we have found that the acceptance rate is even higher, reaching a percentage of 85%. As the projects in this list respect best engineering practices, we expect such a high acceptance rate.

For merged pull requests, an important property is the time required to process and merge them. We found developers, on average, merge a pull request

in 6.3 days. Moreover, teams merge 90% of the pull requests within 10 days, 80% within 3.5, and 50% in the first 6 hours. The pull request data shows that merges take 27% longer than the previous study. Core developers may need more time to assess a pull request because the work volume is higher, due to the increase in the number of pull requests, or because they take more time to ensure proper reviewing. We got similar results for the reporeapers dataset. An observation is that developers take longer to close pull requests. The average closing time is 36.8 days, almost 6 times higher than the mean merge time. In the case of the reporeapers dataset, the time has lowered to 11.2 days, which shows a higher interest in closing the pull requests, no matter the outcome. However, we conclude that developers leave rejected pull requests lingering for a long period of time.

To understand the difference between core team members versus non-core, we compare each group's pull request merge times. For core members, we found the mean merge time is 4.12 days, whereas, for non-core team members, the mean is 13.8 days. By using the *unpaired t test,* we conclude the difference is statistically significant ($p < 0.0001$), meaning that, contrary to what the previous research has shown, a team processes pull requests submitted by core members faster than than those submitted by the external members. Looking at the reporeapers dataset, we have found that the mean merge time for core members is similar, but the one for external developers is almost 20% higher, exceeding the value of 16 days. This shows an even bigger gap between the core team members and external contributors.

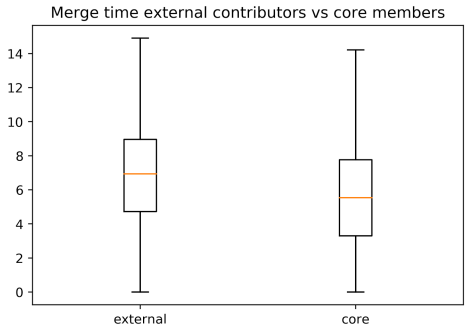

Figure 7. Time to merge pull request for internal vs external team members

Size of pull requests. To understand if the size of a pull requests affects merging, we examine the components of pull request size. A pull request bundles together a set of commits; the average number of commits on a pull request is 3.3 with many containing less than 6 (95% percentile: 7, 90% percentile: 4, 80% percentile: 2). Developers change generally less than 20 files (95% percentile: 25, 90% percentile: 13, 80% percentile: 6), with median number of 8.25. Furthermore, the majority of pull requests contain less than 200 lines of code (95% percentile: 218, 90% percentile: 153, 80% percentile: 86) with a median number of 55. The results show a decrease in value for all parameters compared to the previous paper, meaning that developers prefer to submit smaller pull requests now as compared to 4 years ago. One explanation for this behavior might be that this is a strategy to ensure a higher acceptance rate for the pull requests. Yet, running a Spearman correlation between the merging decision and each parameter did not show any clear correlation. Performing the same analysis on the Reporeapers dataset reveals interesting findings. Although the number of commits and files changed per pull request is double compared to the own dataset, the number of changed lines remains the same. This means that the developers involved in these projects consider the best practice is to commit more often with smaller modifications to the code.

Discussion and code review. A submitted pull request is open for discussion until the team merges it or closes the request. In our dataset, the discussion is usually brief: 95% of pull requests receive less than 12 comments and 80% receive less than 3 comments. Comparing these numbers with the ones from the previous paper, we can conclude that the behaviour of developers has not changed since then. However, as opposed to the previous findings, performing a Spearman correlation reveals that there is a moderate correlation between the number of comments with both the merge decision (rs = 0.31) and the lifetime of the pull request (rs = 0.47). We obtained the same results when using the Reporeapers dataset.

7. FACTORS INFLUENCING MERGING AND MERGE TIME

7.1. GHTorrent dataset

We run the classification processes according to the protocol specified in Section 3. The data comprises a sample of 45,000 pull requests for the merge decision. Out of these, we use 10% to test the resulting model for the machine learning algorithm. Based on the results presented in Table 2, we have selected the *randomforest* classification algorithm for our experiment due to it having the highest area under the curve (AUC) value which represents the

highest accuracy. For the *mergedecision* experiment, *randomforest* achieved an AUC of 0.90.

classifier	AUC
mergedecision	n = 45,000
binlogregr	0.5
svm	0.83
decisiontree	0.84
randomforest	0.9

Table 1. Classifier performance for the merge decision classification task.

To perform the classification, we have used *pyspark*'s implementation of random forests, specifying a number of 100 trees and a maximum depth of 5. Before we execute the algorithm, we filter the list of features to ensure we use features that are independent or not strongly correlated (rs < 0.7).

We have found that *pr_interaction_pr_events* has the biggest importance out of all the features. This means that the most important aspect developers look at when merging pull requests is the number of linked events. *num_participants* is another feature that has a similar importance in the merge decision, meaning when merging a pull request, we take into consideration the number of people participating in pull request discussions. *team_size* is the next feature in order of importance and shows that the merging decision is dependent of the number of core team members that were active during the period the pull request was being reviewed. Also, *num_issue_comments* seems to have a moderate influence on the merge decision. By re-running the same algorithm using only the top 4 features, we have found only a slight decrease in the AUC of the algorithm (0.85 in the case of randomforest)

RQ3: The three main factors that affect the decision to merge a pull request are: ***i)*** *The number of past events linked to it* ***ii)*** *The number of participants in the discussion* ***iii)*** *How big is the core developer team.*

To determine the factors that influence the merge time of pull request, we use a sample of 50000 pull requests and apply the same algorithms. *randomforest* remains the preferred algorithm due to the higher AUC.

classifier	AUC
mergetime	n = 45,000
binlogregr	0.5
svm	0.77
decision tree	0.69
randomforest	0.85

Table 2. Classifier performance for the merge time classification task.

For the *mergetime* task, the dominating feature is more pronounced than in the previous case. We found that *num_participants* is the feature that influences the merge time the most. This feature represents the number of people participating in the pull request discussions, followed closely by *num_issue_comments* (number of discussion comments) and *at_mentions_comments* (number of mentions in the comments section). Judging by these results, we observe that the discussion section has a significant impact on the merge time of the pull requests.

The results are different from the previous paper in most aspects. In the previous paper, the number of comments, the team size and the test coverage were the most influential features for the merging time, whereas our results showed that only the features related to comments had significant impact.

RQ3: The main factors that influence the merging time of pull requests all relate to the discussion section of a PR: i) The number of participants in the discussion section. ii) The number of comments. iii) The number of mentions in the comments.

7.2 RepoReapers dataset

The classification procedure for this dataset is identical to the previous one, except for the size of the sample, which is now reduced to 15,000 pull requests due to time limitations. For this dataset, we obtained similar accuracies of the classifiers for both of the research questions.

We found that the top two features from the first dataset, *prior_interaction_pr_events* and *num_participants,* remain the most influential in this case. The main difference was that the number of discussion comments (*num_issue_comments*) shows a higher influence on the merge decision as compared to the previous dataset. This might mean the community is more influential inside projects that follow RepoReaper's set of engineering

practices. An interesting finding is that the number of *stars* also has a moderate influence on the merge decision. This result represents proof that stars may be a good indicator of the quality of a project.

For the *mergetime* classification task, the two most influential features are the same as the previous dataset (*num_issue_comments* and *num_participants*), whereas the next feature is now the number of pull request review comments (*num_pr_comments).* In spite of the small differences in the final results, the main factors that influence the merging time of pull requests remain the ones related to the discussion section of a pull request.

8. DISCUSSION

8.1 The pull-based development model

Development turnover. Teams often use the pull request development model for faster development turnover, i.e., the time between pull request submission and completion [1, 11, 17]. In the previous study, they found that teams merge the majority of pull requests within the first day with projects showing 80% of pull requests within 4 days, 60% within 1 day, and 30% within 1 hour. Our study yielded longer merge times, on average, 27% higher. We found projects merge 90% of the pull requests within 10 days, 80% within 3.5, and 50% in the first 6 hours. Additionally, we found that the time to close pull requests is six times larger. These numbers indicate that pull requests may have higher turnover now as compared to before, yet, we conclude that developers merge the majority of contributions within the first 24 hours of submission. This finding supports several other works [1, 18, 21].

Managing pull requests. Managing pull requests is an essential part the pull-based development model. Teams use the pull request as a transparent tool to review code to ensure style, quality, and completeness within their projects [20]. Additionally, previous studies show that smaller requests get in more [18, 19]. In our study, we find that team's merge 79.69% of pull requests, while the rest are not. We found the three main factors that affect the decision to merge a pull request are: **i)** The number of past events linked to it **ii)** The number of participants in the discussion **iii)** How big is the core developer team. While, the main factors that influence the merging time relate to the discussion: i) The number of participants in the discussion. ii) The number of comments. iii) The number of mentions in the comments. In the previous study, the authors found that teams close 53% of pull requests because of the distributed nature of processing them [1]. Our advice to teams looking to manage pull requests better is keep your core team review process simple and your merge decision process transparent to contributors.

Attracting contributions. In open source development it is important to attract developers. On Github, the pull request makes it easy for new developers to take part through a mechanism known as "drive-by" commits [22, 24]. As the name suggests, these pull requests typically contain small commits and perform a simple patch. In the previous study, they found 7% of commits come from external developers performing "drive-by" commits [1]. Our study, we did not specifically look into this aspect, but did find that commits from external developers take considerably longer to process, on average 13.8 days as compared to 4.12 days for internal members. These numbers suggest that if projects would like to attract contributors, they must treat outside contributions as they would internal contributions, to improve efficiency and encourage community development.

Crowdsourcing the code review. An important part of a pull request is the code review. In open source development, one of the important reasons to attract contributors is to improve crowdsourcing in code reviews. This is because with more conversation, the quality of code review is likely to be better[21, 25]. In our study, we found, similar to the previous study, that the external member community discussing pull requests is larger than the internal team. This suggests that project teams realize the benefit of outside opinions and external developers are willing to help with the development and review process.

Democratizing development. One of the key findings of this study is that projects do not treat external developers the same as internal developers. This is a discrepancy between our findings and the previous study [1, 23]. In the previous study, there was no clear sign that the distinguishment between core developers and internal developers affected the merge decision or merge time. In our results, we show pull requests from external developers take on average 13.8 days and in the RepoReapers dataset 16 days. These findings show that external developers face a longer approval process. We believe the software community can improve their reviewing processes by looking into better testing, a discussion and code review protocol, and a transparent decision making plan to better accommodate outside contributors.

8.2 Implications

Contributors. Project teams want to improve the success rate of contributions. Our research, similar to the previous study, shows that the majority of pull requests contain less than 200 lines of code and that the number of past events linked to a pull requests best indicates the merge decision. These findings suggest that contributors will achieve the best success if they commit short and sweet, and fix hot areas of code like previous bugs or patch particular features.

Core team. The primary job of the core team is to manage contributions to the project by evaluating all contributor pull requests. To make sure teams process pull requests on time, we recommend, similar to previous studies [1, 15], that development teams invest in a comprehensive testing suite. This allows contributors to validate their proposed changes, and saves developers time reviewing. If a team articulates what they expect in a pull request, the merge time of outside contributions will likely decrease while improving quality.

As suggested by the previous study, a direct application of our results is the construction of tooling to help developers prioritize incoming pull requests, since our data shows, with high accuracy, whether a team will merge a pull request or not. A tool that examines pull request data and history to provide suggestions about where developers should focus could drastically reduce review time. The tool could indicate the code that brings entropy into the system or anomalies that do not match the working project. The tool can provide feedback for contributors submitting pull requests to help them better adhere to project guidelines. Such a tool could prove useful for teams and contributors.

8.3 Threats to validity

Internal validity. Our statistical analysis uses random forests as a way to identify and rank cross-factor importance on two response variables. The results for merge-decision are convincing, and the classification scores for merge-time show considerable improvement over the previous study. Using the two datasets, and a larger set of projects, we believe the validity of our results enhance the previous study.

We extracted data, as did the previous authors, from i) the GHTorrent relational database ii) the GHTorrent raw database iii) each project's Git repository. We found differences in the data abstraction may lead to different results in the following two cases: i) Number of commits in a pull request due to when developers use commit squashing, the system reduces the number of commits to one. ii) Number of files and commits on touched files: commits may contain several files not related to the pull request itself, which in turn affects our results. Therefore, as in the previous study, we filtered out those commits.

External validity. In our study, we merged data from several projects. We treated all projects as equal, even though differences do exist. For example, the largest project in our dataset, as in the previous study, Ruby on Rails, has more than 7,000 pull requests while the smaller ones contain just over 200. While we believe that the uniform treatment of the samples led to more conclusive results in the classification experiment, we may need to look closer at differences among large and small projects. Our random selection cross-validation obtained stable prediction results, this is encouraging showing our dataset achieved accurate results, but further analysis may be necessary.

9. CONCLUSION

The goal of this paper was to understand how the pull-based software development model changed in the last 4 years. We look at the attitude of developers on pull requests and if and how its characteristics (mergetime or acceptance rate) compare to the paper from 2013. To achieve this, we use two datasets as described in Section 4 and a total of approximately 320,000 pull-requests.

Our findings are the following:

1. GitHub registered a growth of 105% in the number of pull requests.The popularity of the pull request development increased, most likely due to the constant growth of Github as a platform.
2. The percentage of merged pull requests increased by 5.2%. Teams merge 90% of the pull requests within 10 days, 80% within 3.5, and 50% in the first 6 hours. Our data shows that merges take 27% longer than the previous study. The average closing time is 36.8 days, almost 6 times higher than the mean merge time. In the case of the Reporeapers dataset, the closing time lowered to 11.2 days. Contrary to what the previous research has shown, a team processes pull requests submitted by core members faster than those submitted by the external members.
3. The three main factors that affect the decision to merge a pull request are: **i)** The number of past events linked to it **ii)** The number of participants in the discussion **iii)** How big is the core developer team. The main factors that influence the

merging time of pull requests all relate to the discussion section of a PR: i) The number of participants in the discussion section. ii) The number of comments. iii) The number of mentions in the comments.

Beside these findings, our analysis registered the following implications:

1. To assess how the popularity of pull request projects is changing, future research is necessary to understand if the number of pull requests to commits declines, levels off, or is an anomaly.
2. To determine how developers use the pull based software development, research to discover new features may reveal trends within the community.
3. To answer the fourth research question, not addressed in this paper, new work could use natural language processing algorithms or other methods to compare and identify factors that contribute to the rejection of a pull-request.

The dataset used in this study, we extracted using the criteria described in Section 4 from the *pullreqs* project found on GitHub at *gousiosg/pullreqs*. We used the Jupyter Notebook from the previous study to perform all algorithms and get the results for our analysis. The analysis and datasets for our study is on our Github repository, *alexanderblnf/pullreqs-feature-analysis*. Using this information, any interested researcher can replicate the study.

References

[1] G. Gousios, M. Pinzger, A. van Deursen. An Exploratory Study of the Pull-based Software Development Model.

[2] GitHub. (2017). The State of the Octoverse 2017. Github Octoverse. Retrieved on Nov. 3 from: https://octoverse.github.com/

[3] S. Chacon. Pro Git. Expert's Voice in SoftwareDevelopment. Apress, 1rst edition, Aug 2009.

[4] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2014, May). The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories* (pp. 92-101). ACM.

[5] Lanubile, F., Ebert, C., Prikladnicki, R., & Vizcaíno, A. (2010). Collaboration tools for global software engineering. *IEEE software*, *27*(2).

[6] Collins-Sussman, B., Fitzpatrick, B. W., & Pilato, C. M. (2002). Version control with subversion. *Version Control with Subversion*.

[7] Google. (2017). Comparisons of versioning systems. Google Trends. Retrieved from. https://trends.google.com/trends/explore?date=all&q=%2Fm%2F05vqwg,%2Fm%2F012ct9,%2Fm%2F08441_,%2Fm%2F08w6d6,%2Fm%2F09d6g&hl=en-US&tz=&tz=

[8] Firestine, B. (2017). Celebrating nine years of GitHub. GitHub. Retrieved from: https://github.com/blog/2345-celebrating-nine-years-of-github-with-an-anniversary-sale

[9] Atlassian. (2017). Making a pull request. Atlassian Git Tutorials. Retrieved from: https://www.atlassian.com/git/tutorials/making-a-pull-request

[10] Munaiah, N., Kroh, S., Cabrey, C., & Nagappan, M. (2017). Curating GitHub for engineered software projects. *Empirical Software Engineering*, *22*(6), 3219-3253.

[11] K. Peterson. The github open source development process. Technical report, Mayo Clinic, May 2013.

[12] N. McDonald and S. Goggins. Performance and participation in open source software on github. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '13, pages 139–144. ACM, 2013.

[13] E. Shihab, C. Bird, and T. Zimmermann. The effect of branching strategies on software quality. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '12, pages 301–310. ACM, 2012.

[14] Gousios, G., & Spinellis, D. (2012, June). GHTorrent: GitHub's data from a firehose. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (pp. 12-21). IEEE Press.

[15] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the Apache server. In Proceedings of the 30th international conference on Software engineering, ICSE '08, pages 541–550. ACM,2008.

[16] G. Gousios. The GHTorrent dataset and tool suite. In MSR '13: Proceedings of the 10th Working Conference on Mining Software Repositories, May 2013.

[17] O. Baysal, R. Holmes, and M. W. Godfrey. Mining usage data and development artifacts. In 9th IEEE Working Conference on Mining Software Repositories (MSR), 2012, pages 98–107. IEEE, 2012.

[18] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the Apache server. In Proceedings of the 30th international conference on Software engineering, ICSE '08, pages 541–550. ACM, 2008.

[19] P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In MSR '08: Proceedings of the 2008 international working conference on Mining software repositories, pages 67–76. ACM, 2008.

[20] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Leveraging transparency. IEEE Software, 30(1):37–43, 2013.

[21] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. ACM Trans. Softw. Eng. Methodol., 11(3):309–346, 2002.

[22]Pham, R., Singer, L., & Schneider, K. (2013, May). Building test suites in social coding sites by leveraging drive-by commits. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 1209-1212). IEEE Press.

[23] C. Jensen and W. Scacchi. Role migration and advancement processes in OSSD projects: A comparative case study. In ICSE '07: Proceedings of the 29th international conference on Software Engineering, pages 364–374. IEEE Computer Society, 2007.

[24] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider. Creating a shared understanding of testing culture on a social coding site. In Proceedings of the 2013 International Conference on Software Engineering, ICSE'13, pages 112–121. IEEE Press, 2013.

[25] P. C. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. University of Victoria, Canada, Tech. Rep. DCS-305-IR, 2006.